The Hyperdimensional Stack Machine

Thomas Yerxa[1], **Alexander Anderson**[1,2], **Eric Weiss**[2]
1. University of California, Berkeley Department of Physics, 2. Redwood Center for Theoretical Neuroscience

Summary. This work outlines the use of hyperdimensional (HD) computing to design continuous versions of discrete data and control structures. In HD computing everything is represented by a very high dimensional vector, rather than by a component of a lower dimensional vector. The highly distributed nature of this representation makes for components that are highly robust to noise and other corruptions. We tested these structures using two simple tasks: string reversal and the execution of Reverse Polish Notation sequences.

Highly distributed representations tend to be very interpretable, highly fault tolerant, and robust against large amounts of additive noise. These properties are useful for practical reasons and also important to understand from a computational neuroscience perspective because the degree of robustness in very noisy environments is one of the key differences between computations in artificial systems and those that take place in the brain. In this work we develop a method for using hyperdimensional computing to develop distributed representations with the above properties. In very high dimensions vectors that are randomly sampled are very likely to be very nearly orthogonal to each other. This key fact allows for the construction of Vector Symbolic Architectures (VSA), which use high dimensional "hypervectors," as atomic units rather than individual elements of a low dimensional vector; in this way VSA's are highly distributed approximate computing systems. Additionally, operations in this framework are continuous and differentiable which makes them compatible with standard machine learning techniques.

Three operations on hypervectors allow for symbolic computation: superposition, element-wise multiplication, and permutation. Superposition of two hypervectors effectively associates the pair in an unordered set. Element-wise multiplication of two vectors results in a third vector that represents the bound pair of the two, and unbinding can be achieved via element-wise multiplication with the multiplicative inverse of one of the original vectors. In this way binding allows for the composition of two symbols. Finally, permuting the elements of a hypervector results in a new vector that is dissimilar to the original. Permutations are invertible and multiple applications of a single permutation may be be chained together which means that permutations can be used to index vectors (the index being the power of the permutation that has been applied to it). This work uses vectors whose components are random unit-norm complex numbers, so each vector's multiplicative inverse is simply its conjugate.

Consider first the construction of a hyperdimensional stack, which can be achieved with a single hypervector. A stack is a last in first out data structure defined by two operations: push and pop. The push operation puts an object in the front of a queue, and the pop operation removes an object from the front of a queue. If $\vec{\mathbf{m}}$ is a stack vector, a vector $\vec{\mathbf{r}}$ can be pushed onto the stack by the operation $\vec{\mathbf{m}'} = \vec{\mathbf{r}} + \rho\vec{\mathbf{m}}$, where ρ is a random permutation characteristic of the stack. Then if $\mathbf{V}$ is defined to be a matrix whose rows are the hypervector embeddings of the symbols being manipulated, popping from the stack is defined by $\vec{\mathbf{r}}_{popped} = \mathbf{V} \cdot softmax(\vec{\mathbf{m}} \cdot \mathbf{V}^\dagger)$, $\vec{\mathbf{m}'} = \rho^{-1}(\vec{\mathbf{m}} - \vec{\mathbf{r}}_{popped})$ (here $\mathbf{V}^\dagger$ is the conjugate transpose of $\mathbf{V}$). The procedure for obtaining $\vec{\mathbf{r}}_{popped}$ is sometimes referred to as a cleanup and can be written more compactly as $CleanUp(\vec{\mathbf{m}}, \mathbf{V})$. The first matrix multiplication results in the components of each symbol on the top layer of the stack, then the softmax converts these components into a probability distribution; the softmax largely removes the noise incurred by the imperfection of the orthogonality approximation. Finally the "cleaned up," weights are used to re-embed in the HD space. In this distributed implementation the push and pop operations change the power of ρ associated with each term in the sum $\Sigma_{i=0}\rho^i\vec{\mathbf{v}}_i$, and the object at the front of queue is the $i = 0$ term.

Next we define an HD finite state machine (FSM). An FSM is defined by a transition table that maps a state and an input to a new state and an output, we can represent this structure with two hypervectors. Let $\mathbf{V}$ again be a matrix of symbol embeddings, $\mathbf{S}$ be a matrix of state embeddings, and $\mathbf{A}$ a matrix of output embeddings; let each of these matrices have rows denoted by the vectors $\vec{\mathbf{v}}_i$, $\vec{\mathbf{s}}_i$, and $\vec{\mathbf{a}}_i$ respectively. Then we can define two vectors $\vec{\mathbf{T}} = \Sigma_{i,j}\vec{\mathbf{v}}_i * \vec{\mathbf{s}}_j * T^{HD}(\vec{\mathbf{v}}_i, \vec{\mathbf{s}}_j)$ and $\vec{\mathbf{U}} = \Sigma_{i,j}\vec{\mathbf{v}}_i * \vec{\mathbf{s}}_j * U^{HD}(\vec{\mathbf{v}}_i, \vec{\mathbf{s}}_j)$ where T^{HD} is a function $T^{HD} : (\vec{\mathbf{v}} \in \mathbf{V}, \vec{\mathbf{s}} \in \mathbf{S}) \rightarrow \vec{\mathbf{s}} \in \mathbf{S}$ and U^{HD} is a function $U^{HD} : (\vec{\mathbf{v}} \in \mathbf{V}, \vec{\mathbf{s}} \in \mathbf{S}) \rightarrow \vec{\mathbf{a}} \in \mathbf{A}$; these two functions define the transition table of the FSM. The transition between states is then determined by $\vec{\mathbf{s}}_{new} = \vec{\mathbf{s}}^*_{current} * \vec{\mathbf{v}}^*_{input} * \vec{\mathbf{T}} \approx \vec{\mathbf{1}} * \vec{\mathbf{1}} * T^{HD}(\vec{\mathbf{v}}_{input}, \vec{\mathbf{s}}_{current})$, and an identical operation determines the output of the FSM. Because the vectors are high dimensional, the other terms present in $\tilde{\mathbf{s}}_{\mathbf{new}}$ will all be orthogonal to all vectors of interest, and thus can be effectively treated as noise. The unbinding operations "reveal," the output of the transition functions which set the next state and output.

From here it is straightforward to design a system that executes a stack algorithm. The procedure for selecting what action will be taken is already defined, but a system using these structures must also

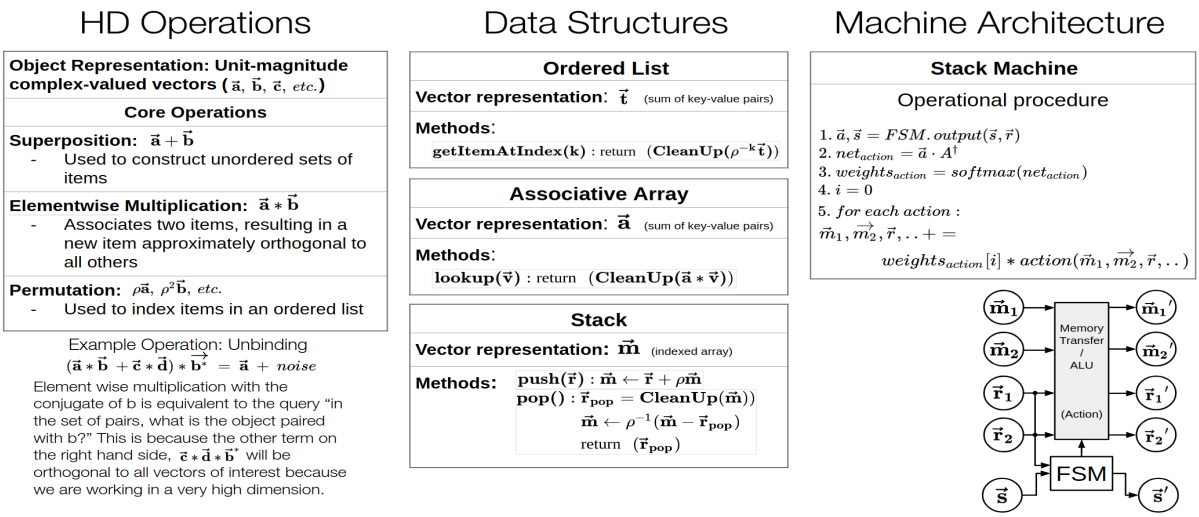

Figure 1: The first two columns contain summaries of some of the important implementations and abstractions discussed so far. The third describes and visualizes how these can be used to design an HD stack machine. Tasks are executed by iterating the pseudo-code shown for some number of time steps.

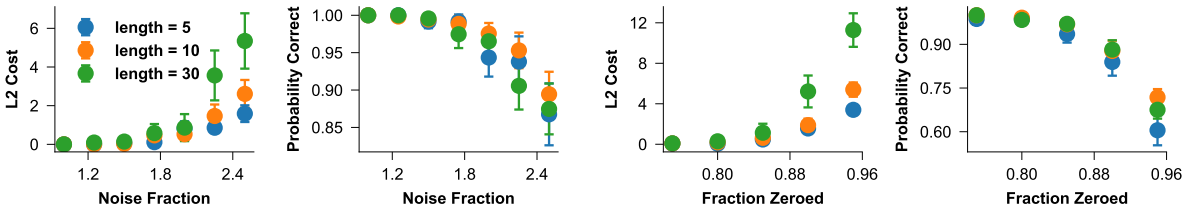

Figure 2: These plots show how a system performing a string reversal task (on sequences of three different lengths) responds when the vectors that define the FSM are corrupted. Because the underlying representations are vectors we can characterize the effect by both $||\vec{m}_{ideal} - \vec{m}_{corrupted}||^2$ which is the 'L2 Cost,' and the probability of reading out the correct element in the output (which is the more familiar metric for sequence to sequence tasks). The left two plots show the effect of white noise being added, 'Noise Fraction' is defined to be the standard deviation of the Gaussian noise added to each component divided by the mean value of each component ($\frac{\sigma_{noise}}{\mu_{vector}}$). The right two plots show the response when a random set of the components of each vector are set to zero. The structures are highly robust, as there is little to no cost associated with a noise fraction below 1.2, or the complete failure (zeroing out) of 80% of the components. The dimensionality of the FSM vectors was 1,000 for these calculations.

define how each action affects each component (stacks, registers, etc.). Obvious choices include popping from either stack and placing the result on one of the registers and pushing a register vector onto the output stack, but others can be task specific. The HD calculator requires actions that execute arithmetic operations on the symbolic values of the registers. These actions are accomplished by feeding the low dimensional representations of the registers $\vec{r}_1 \cdot \mathbf{V}^\dagger$ and $\vec{r}_2 \cdot \mathbf{V}^\dagger$ into an arithmetic logic unit (ALU).

In addition to being both interpretable and robust, each of the operations that make up a VSA is differentiable and are thus readily compatible with machine learning schemes that use gradient descent. For example program synthesis could be achieved by initializing the vectors that define the FSM to random values, and optimizing them with standard methods. This framework is rich enough that is easy to imagine using a VSA to develop other data structures, such as linked lists or trees, that could be put to use in increasingly complicated tasks. These structures have a host of potential applications and properties that are important to investigate to better understand the differences between artificial and neural computation.

References

[1] P. Kanerva,"Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," Cognitive Computation 1.2 (2009): 139-159.

[2] T. Plate, "Holographic reduced representations," IEEE Transactions on Neural networks 6.3 (1995): 623-641.